An Implementation of CuPIDS: Evaluating the Effectiveness of Multi-processor Information System Security

Blinded

Abstract

The Co-Processing Intrusion Detection System (CuPIDS) project explores practical improvements in information system security and survivability through dedicating computational resources to system security tasks in a shared resource, multi-processor (MP) architecture. Our research explores ways in which this architecture offers improvements over the traditional uni-processor (UP) model of security. This paper describes the details of an implementation of such a system. This prototype is used to validate our research theses and explore some of the performance benefits and tradeoffs of dedicating computational resources to computational tasks as well as the cost of creating and using such a system.

1. Introduction and Philosophy

Our primary research thesis for CuPIDS is that a tightly focused, parallel monitoring process can detect illegitimate behavior more quickly than can a process operating in either scheduler-interleaved or interposing mode. Our experimental results thus far provide solid support for that thesis; however, transforming the theory into an architecture, and the architecture into a prototype which allows us to explore the validity of the theory posed a number of challenges. These difficulties included efficiency related concerns, host operating system and application integration, and performance measurements. The efficiency concerns include both system runtime performance and the increased workload on the application developer imposed by the CuPIDS architecture.

A cornerstone of the CuPIDS philosophy is that security is more important than raw performance—particularly with regards to mission critical applications and servers—we also recognize that in order for this research to be useful it must operate without high performance costs as perceived by system users and developers. For the former group we explore the use of hardware to accelerate security tasks which have primarily been handled by software. Examples include enforced parallel security monitoring and exploration of ways in which system events can be efficiently generated and moved through the CuPIDS architecture. This experimentation helped us define how CuPIDS integrates into the operating system and applications it protects. Realizing some of our goals, such as complete visibility into the runtime state of a protected process, required the addition of a CuPIDS specific application programming interface (API) into both the operating system and system runtime libraries as well as the protected process itself. Finally, demonstrating the efficacy of our architecture required that we measure how long it takes CuPIDS to detect and respond to an illegitimate event and compare this result to those produced by more conventional architectures. The simultaneous nature of CuPIDS' operations posed challenges in this regard. Further we needed to measure the performance impact the architecture imposes on a realistic system. The implementation process allowed us to evaluate the likely costs a developer of a CuPIDS application will incur.

CuPIDS represents a paradigm shift in information system security—one in which we shift away from the standard uni-processor intrusion detection (StUPIDS) model used by most architectures in this domain to a multi-processor model in which highly focused monitoring tasks run simultaneously with the process they protect. Our results uncover gains in detection speed while incurring reasonable and minimal overhead costs. We demonstrate that running concurrently with attack code affords CuPIDS opportunities to detect and respond to attacks that are not available to StUPIDS. Additionally, because the opportunity exists to detect attacks while they occur without waiting for a context-switching event (either between user processes or between user and kernel mode) CuPIDS is able to respond more quickly and attacks are detected in realtime[1] and with high fidelity. These results represent advantages that are difficult or impossible to achieve on a uni-processor system—no matter how powerful.

The focus of this paper is on the tension between usability and security. We evaluate of the psychological acceptability [2] of the CuPIDS architecture—does it as a security mechanism make the system as a resource more difficult to utilize or less efficient, and if so by how much? This paper describes an implementation of CuPIDS in which we briefly discuss the origins and architecture, and then delve into what we have learned in building a working prototype

1 As defined by Kuperman in [1]

and what the runtime and developmental costs are likely to be. Using commodity hardware and lightly customized commodity software we demonstrate that the performance impact of the CuPIDS architecture is modest and tolerable.

2. Background

The motivation and architecture of CuPIDS is discussed in [citation blinded] and initial performance results as well as discussion of CuPIDS ability to perform self-protection and self healing are presented in [citation blinded].

We assume the reader is familiar with the Intrusion Detection System (IDS) literature and the threat model that body of research addresses. Axelsson's in-depth, thorough taxonomy and survey of the field of intrusion detection in 2000 is a good starting point for those unfamiliar with the field [3]. We draw from those techniques and augment them in ways that make use of the MP paradigm. Many of the specific intrusion detection techniques a CuPIDS will use differ from their StUPIDS counterparts only in the real-time, simultaneous monitoring nature of their use. As intimated by its name CuPIDS is currently focused on intrusion detection (ID); however, the ideas embodied by the architecture are general in nature. We anticipate these results will be directly useful in the related domains of software debugging and computer forensics, and we drew inspiration and ideas from these communities while exploring the ideas which turned into CuPIDS.

We are concerned with a general threat model that assumes:

- Processes running at any privilege level in the production parts of the system may be compromised at any time after boot is complete.
- Attacks may come from local or external users or a combination of both.
- Attacks may succeed without ever causing a context switching event.

The current CuPIDS prototype implements a form of runtime invariant testing similar to that described by Patil and Fischer [4]. They discuss how including runtime error checking may slow applications by as much as a factor of 10. This is too expensive for most application; therefore once debugging and testing is complete, runtime error checks are disabled or removed. The authors responded by creating guard programs that model the execution of the production program—including the runtime error checks—but only at the pointer and array access level. Offline, interleaved, and parallel versions of this architecture were discussed. We use the idea of exporting runtime checks to a shadowing process; however, our work differs from theirs in that we focus on real-time monitoring of the actual memory locations in use by the production process as well as a much larger set of monitoring capabilities.

CuPIDS is similar to the work done in external modeling of a process' activity. Examples of such work include the research of Haizhi Xu et al. in using context-sensitive monitoring of process control flows to detect errors[5]. They define a series of "waypoints" as those points along a normal flow of execution where an application dips into the system call interface. They demonstrated good results in detecting attempts to access system resources by a subverted process. CuPIDS makes use of a similar idea to their waypoints in its checkpoints, those points in both the interactive and passive systems where CuPIDS is notified of events in which it is interested; however, CuPIDS checkpoints are much finer-grained and are generated within the production process as well as its interaction with the external environment. As an example, CuPIDS uses function call entry and exit information to perform rough granularity program counter tracking and validation as well as model a program stack for use in detecting illegitimate control flows within a process code segment. Other closely related work includes that done by Feng et al. [6]. The authors explored extracting return addresses from the call stack and using abstract execution path checking between pairs of points to detect attacks. Finally, Gopalakrishna et al. [7] present good results in performing online flow- and context-sensitive modeling of program behavior. Gopalakrishna's Inlined Automaton Model (IAM) addresses inefficiencies in earlier context-sensitive models [8, 9] by using inlined function call nodes to dramatically reduce the non-determinism in their model while applying compaction techniques to reduce the model's memory usage. Using an event stream generated by library call interpositioning, IAM is shown to be efficient and scalable even in a StUPIDS architecture. The techniques used by IAM fit naturally into the CuPIDS architecture. All of these models can be run as CuPIDS shadow processes (CSP)[2]. The model simulations can be run as CSPs, each getting its inputs from the CuPIDS event streams.

Offloading security work onto co-processor has been explored by research such as that discussed in [10, 11, 12, 13, 14]. An example of the latter category includes the work done by Zhang, et al. in describing how a crypto co-processor is used to perform some host-based intrusion detection tasks[13]. That research examines the possibility of using hardware designed for securely booting the system to run an intrusion detection system. The benefits from doing so include protecting the IDS processor from the production processor, and offloading IDS work from the main processor onto one dedicated for that task—goals CuPIDS shares. Strengths of this approach include high attack resistance for code running in the co-processor system. Draw-

2 Described in Section 3

backs of the approach include the lack of ready visibility into the actions of the main processor and operating system. These strengths and drawbacks also exist in the use of virtual machine architectures.

Garfinkel and Rosenblum discuss a novel approach to protecting IDS components [15]. Here the the primary goal of enhancing attack resistance is met by isolating the IDS in virtual machine monitor (VMM). The VMM approach has much in common with the reference monitor work discussed by Anderson [16] and Lipton [17] in that it provides a means by which the IDS can mediate access between software running in the virtual host and the hardware. It can also interpose at the architecture interface, which yields a better view into the system operation by providing visibility into both software and hardware events. A traditional software-only IDS does not have this advantage. Of course, the IDS running in the VMM has visibility only into hardware-level state. This means that the IDS can see physical pages and hardware registers, but must be able to determine what meaning the host O/S is placing on those hardware items. By running as part of the host O/S, CuPIDS maintains complete visibility into the software state of the entire system, but currently lacks the protection afforded VMs and secure co-processor architectures. Future work on CuPIDS will use hardware protection mechanisms such as those in the Intel IA32 [18] processor line to provide protection of security specific components as well as critical operating system components.

3. CuPIDS Architecture

The CuPIDS architecture was initially presented in [citation blinded], and is summarized here.

3.1. High Level Design

CuPIDS operates using the facilities and capabilities afforded by a general purpose symmetrical multi-processing (SMP) computer architecture. Common operating systems such as Windows, Linux, and FreeBSD running on SMP architectures use the CPUs symmetrically, attempting to allocate tasks equally across the CPUs based upon system loading [19]. CuPIDS differs from these architectures in that at any point in time one or more of the CPUs in a system are used exclusively for security related tasks. This asymmetrical use of processors in a SMP architecture is a significant departure from normal computing models, and represents a shift in priority from performance, where as many CPU cycles as possible are used for production tasks, to security where a significant portion of the CPU cycles available in a system are dedicated solely to protective work. One possible CuPIDS software architecture is depicted in Figure 1. The dark components represent production tasks and services and run on one CPU while the light components represent the CuPIDS monitors and run on a separate CPU. The regions of overlap depict CuPIDS ability to monitor the resource usage of production components.

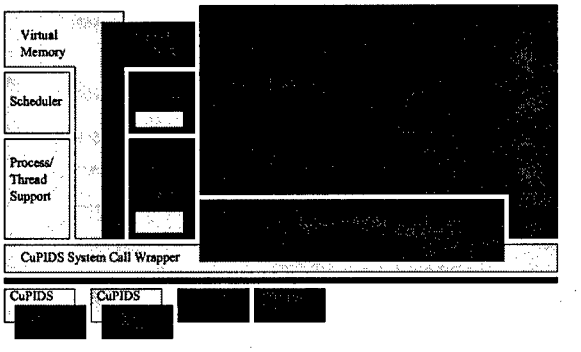

Figure 1. Basic software architecture

The operating system as well as user processes are divided into components that are intended to run on separate CPUs. The intent behind this separation is twofold: performance, where we seek to minimize the runtime penalty imposed by the security system, and protection, where we are concerned with the completeness of detection. By ensuring the processes responsible for detecting bad events are actively monitoring the system during periods in which bad events can occur—the CuPIDS architecture requires that when a CPP is executing its associated CSP is also on a CPU—we provide a real-time detection capability (using Kuperman's definition). The system protection derives in part from the ability to detect bad events as they occur but before the results of these events can cause a system compromise.

A program intended to operate in CuPIDS is divided into two components, a CuPIDS monitored production process (CPP) and a shadowing CuPIDS process (CSP) as depicted in Figure 2.

As the figure shows, CuPIDS processes differ from the traditional process paradigm in the asymmetric sharing of memory between the CSP and CPP. The CPP is a normal process and contains the code and data structures that are used to accomplish the tasks for which the program is designed. It may also contain code and data structures with which information about the state of the running process is communicated to the security component. In addition to the normal process code and data structures, the CSP's virtual memory is modified to contain portions of the CPP's virtual memory space (depicted in the figure as Shadow Memory). This allows the CSP to directly monitor the activities of the production component as it executes.

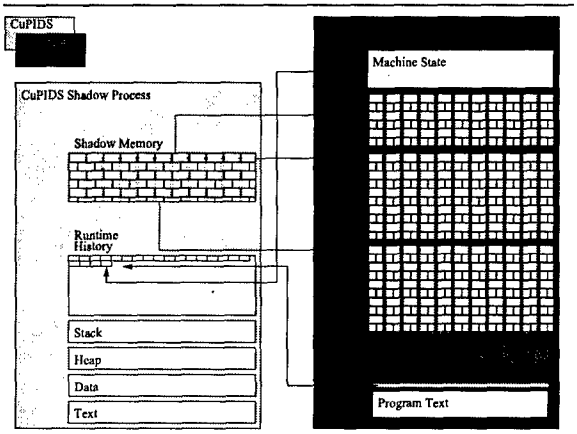

Figure 2. CSP and CPP details

Our initial work assumes the CPP developer is aware of CuPIDS and the CPP communicates its state to the CSP by sending a stream of messages about events of interest to CuPIDS. Later work will investigate what types of real-time monitoring are possible for uninstrumented applications.

3.2. Protective Activities

The CuPIDS architecture currently supports three types of protective activities: Application startup/shutdown validation, state monitoring, including invariant testing, and execution monitoring.

Application startup/shutdown: Startup tasks include verifying the authenticity of both the CSP and CPP as well as any supporting configuration files. The CSP is loaded and started executing. It then loads the CPP into memory, establishes any needed hooks into the CPP's VM space, initializes the various event communication systems, and finally starts the CPP running. Shutdown tasks include verifying that the CPP shutdown path followed a legitimate code path. Additionally, any runtime history data is saved to disk.

State monitoring & Assertion verification: The enhanced CuPIDS API in the kernel allows a CSP to monitor nearly all aspects of the CPP's operating environment and state, including its entire VM space and any related kernel data structures and excluding only the internal processor state while the CPP is on a CPU. One use of this capability is invariant testing. Invariant testing is a two stage process involving pre-compilation work and runtime invariant checking. The pre-compilation task involves determining which variables need monitoring, defining invariants for those variables and exporting that information in a form that can be used by the CSP. The compiler is also used to automatically instrument the CPP by adding event generation hooks into each function prologue and epilogue. Invariants are currently snippets of code that could be directly included in the CPP's code (similar to the run-time debugging tests discussed earlier). They are compiled into the CSP's code, and when one is used, it is given appropriate pointers into the CPP's virtual memory space and executed. Currently these are manually written; however, work is underway to allow a programmer to indicate, via pragmas, to the compiler that a particular variable needs protection and the compiler will automatically generate the invariant testing code in the CSP.

Runtime execution monitoring Runtime monitoring includes a number of activities and capabilities that give the CSP visibility into the operation of the CPP. An example includes generating events so the CSP is made aware of the creation, accesses to, and deletion of a protected variable's lifespan. Other events export an execution trace to CuPIDS via function call monitoring, and interactions between the CPP and external environmental entities such as calls to runtime libraries and the operating system. Call monitoring consists of the CPP sending a stream of function/library/system call entry and exit events to the CSP. The CSP then uses a model based upon how the CPP is supposed to operate to verify if that stream is legitimate.

In addition to the direct monitoring of the CPP performed by the CSP, CuPIDS has a number of background capabilities that augment the CSP's capabilities. These include the ability to intercept and direct low-level system activities such as interrupts and signals, controlling the system scheduler to enforce the segregation of the CuPIDS and system CPUs and ensuring that whenever a CSP is chosen to run, its associated CSP is also placed on the CuPIDS' CPU. Additionally, CuPIDS provides a streamlined, interrupt-based, communication interface for moving event records from the CPP to the CSP running on a different CPU.

3.3. Self-healing/Self-protection

There are a number of well-known-to-be-dangerous library and syscalls [20]. Among the most common exploits publicly available are buffer overflows that use unsafe string handling library functions to overflow vulnerable buffers. Using a combination of stack modeling, library call event monitoring and the virtual memory mapping capability it is possible for CuPIDS to automatically detect and generate

detection signatures for certain common classes of vulnerabilities such as stack-based overflows. In many cases buffer overflows use known library function such as strcopy(3). When CuPIDS is notified of a call to strcopy it can create a copy on write (COW) mapping of the page(s) containing the buffer and surrounding memory region. If information about buffer sizes is available to the CSP, either automatically generated or inserted by the programmer in the form of CuPIDS memory operation events it becomes possible for CuPIDS to not only detect and generate signatures for anomalous events, but also to recover from them automatically. It does so by using the saved copy of stack (or heap) pages to recreate the process' memory state as it was before the overflow, and copying only the correct amount of data into the buffer from the corrupted pages. While in the case of an exploit attempt the data ending up in the buffer may not be what the CPP programmer intended, the overall effect to the program is the same as if a safe string copy function such as strncopy(3) had been used. In addition, error variables or signals may be set to indicate that something unexpected occurred.

4. Implementation

We have implemented a prototype CuPIDS. This section briefly describes the current state of that prototype. Our experimentation uses FreeBSD, currently 5.3-RELEASE [21]. We have added to the operating system API a set of CuPIDS-specific system calls that give CuPIDS processes visibility into and control over the execution of a CPP. Examples of the new functionality include the ability to map an arbitrary portion of the PP's address space into the address space of a CSP, a means by which signals destined for and some interrupts caused by the CPP are routed to the monitoring CSP, etc. The operating system kernel has been modified to perform the simultaneous task switching of CPPs and CSPs, a CSP protected loading capability as discussed above in section 3.2, and hooks into various kernel data structures have been added to allow the CSP better visibility into CPP operation and for runtime history data gathering.

Our experimentation to date has focused on protecting specific applications[3]. We perform interactive monitoring based upon automatically generated instrumentation from the compiler as well as CPP programmer defined invariants for key variables. CuPIDS has the capability to examine program binaries and extract explicit white-lists about which system resources are used by the CPP, and then save this information in a form usable by the CSP. As the CPP runs it sends messages to the CSP notifying it about operational activities such as protected variable lifetime events (creation, accesses and deletion) as well as control flow events (currently all function call entry and exits, to include library and syscall invocations) are passed to the CSP as well. The CSP receives these messages and uses them to ensure the CPP is operating correctly. In the case of variables the CSP performs pre- and post-condition invariant checking, and in the case of flow control, it verifies that all function calls are to and from legitimate locations within the CPP text segment. It also maintains a model of the CPP call stack and verifies all function returns are to the correct locations, etc.

4.1. Basic Capabilities

The CuPIDS architecture defines a number of capabilities that are used to perform ID.

4.1.1. Event Communication Interfaces We investigated how best to move information around the system, and selected two communications mechanisms to explore. The first mechanism is based upon the syscall-based SysV message interface. Using this interface a CPP creates a message and sends it to the CSP using msgsnd(3). The CSP receives the message using msgrcv(3) and processes it. This mechanism works well for low throughput traffic types of activities such as focused invariant monitoring; however, it does not perform well when large numbers of events need to be communicated between processes. To facilitate high throughput needs such as that imposed by instrumenting all function, library, and system calls we developed a high-speed, low-drag interrupt-based communications mechanism. We observed that much of the cost in the SysV interface comes from the syscall interface. A great deal of work is done after the kernel is invoked simply to validate the syscall and route it appropriately. The message interface itself is also quite complex and that complexity adds more time costs to each invocation of msgsnd or msgrcv. The combined effect of these result in the execution of several thousand machine instructions for each syscall. We designed a software interrupt-based mechanism for moving events between CPP and a kernel buffer that only requires about 40 machine instructions, thus eliminating much of the overhead imposed upon the CPP by the SysV IPC. Section 5.2 describes the performance gains realized by this mechanism.

We should note that we did not use a shared memory message passing interface because of concerns that a compromised CPP could easily defeat the CSP using spoofed communications. By forcing the communications to go through the kernel we gain the ability to verify the authenticity of events[4].

3 The techniques involved are largely applicable to operating system protection as well

4.1.2. Memory Mapping The CuPIDS architecture requires that the CSP be able to directly monitor the virtual memory space of the CPP. We added to FreeBSD a system call that takes an address in the CPP process VM space and maps that address into the CSP VM space. The mapping is asymmetrical in that the CSP is aware of the shared memory but the CPP is not. The normal set of protections can be applied to a mapping, but thus far we have not discovered a need for anything other than CSP having read and write access to the memory. Some of the self-healing and forensics tasks require the ability to save old state, thus we provide an ability to make copy on write (COW) mappings of memory locations. This allows for efficient copying of modified pages using the kernel's existing and efficient VM manipulation mechanisms.

4.1.3. Interrupt Routing Hardware signals and interrupts can be intercepted and routed to the CSP for handling. This includes write protection faults for CPP, CSP, or kernel text segments. This mechanism allows the CSP to detect attempts by a process to modify its own code; this is true even in the case where some mechanism such as a buffer overflow brought in code that sets up signal handlers to catch this kind of fault. This ability of CuPIDS to validate system events that could lead to compromise is one of its key strengths. In this capacity it is acting as a reference validation mechanism [16].

4.1.4. Event Types There are a number of possible events generated in the CPP and in the kernel in response to CPP activity.

Startup/Shutdown: The CuPIDS architecture requires that when a CuPIDS production process is executing (actually running on a production CPU), that its associated protective process is executing on a CuPIDS CPU.

When the operating system is instructed to execute a protected program it first validates the integrity of both the production and protective programs using a pre-computed cryptographic signature or some other mechanism. If both programs are valid, the O/S first loads the security process into memory, then the production process, and starts the security process executing on a security CPU. The security process establishes any hooks it needs into the production process' memory space and operating environment (wrappers around library and system calls, etc.). When the security process is ready the O/S starts the production process running on a production CPU. As the production process is switched onto and off of the production CPUs the operating system ensures the protective security process is always running whenever the production process is running.

Variable Creation/Use/Deletion: The flowchart in Figure 3 depicts how CuPIDS performs interactive monitoring of a protected variable. Here the protective process is notified of the production process' entry into a region in which a watched variable may be modified. This notification may come from instrumentation embedded in the production process, or it may result from the protective process setting a tripwire in the instruction stream of the production process or on the variable's memory location. The pseudo code illustrated in Figure 4 shows examples of the variable protection instrumentation embedded in the CPP (the $CuPIDS\_var\ldots$ calls invoke the CSP notification mechanism), while Figure 5 depicts the actions taken by the CSP upon notification that variable access is complete.

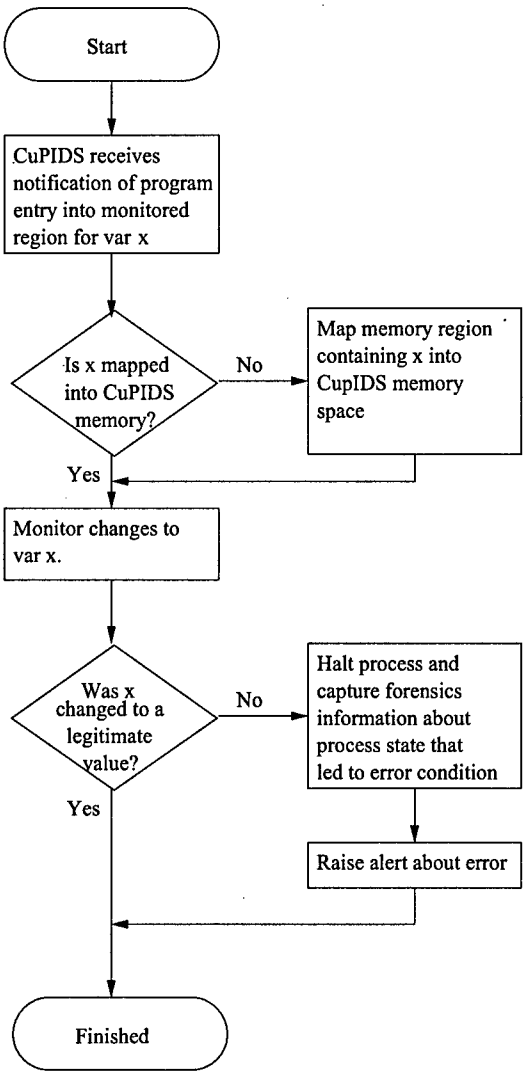

Figure 3. Variable Protection Flowchart

Ideally, the programmer creating and using the variable

4 While this does assume the kernel has not been compromised we can use CuPIDS-like techniques to protect the kernel as well

```
...
CuPIDS_var_create(varID=0,
   var_address = &protected_var);
...
CuPIDS_var_access_begin(varID=0,
   var_address = &protected_var);
cin >> protected_var;
CuPIDS_var_access_end(varID=0,
   var_address = &protected_var);
...
CuPIDS_var_delete(varID=0);
...
```

Figure 4. CPP Variable Protection Code

```
bool CheckVar0PostCondition(void *var){
  if(*var > 42 || *var < 21)
    return false
  else
    return true
} ...
//Msg access end handler
if(varID = 0)
 if(!CheckVar0PostCondition(var_address))
    RaiseAlarm();
...
```

Figure 5. CSP Variable Protection Code

knows what values the variable can legitimately take on; these values are used by CuPIDS as pre- and post-condition invariant tests used to validate the changes or attempted changes to a variable. Other inputs are possible. For example, the size of a buffer is known when it is created, and this information can be used by the protective process to determine if data placed into a buffer overruns the ends of the buffer. If the changes to the variable were legitimate, the production process is allowed to continue execution. If not, the protective process will take some action ranging from annotating the problem in a log to halting the production process or potentially the entire system. In any case, it will likely capture forensics information about the state of the production system leading to the erroneous value being entered into the variable and the changes that took place.

4.1.5. White Lists White List Creation: Currently the white list is created by parsing the program's binary file and extracting function, library, and system call source/destination pairs and storing them in files. This process is not precise; there are complications such as function pointers and occasional calls by non-CSP protected helper programs and libraries. Currently CuPIDS has the ability to augment the white list with call pair events that are not in the white list but that occur during training sessions. Given some cleverness it may be possible to construct a complete white list directly from the binary without requiring training runs; however, it is simpler to handle it this way for our initial prototype.

These files are parsed by the CSP during initialization and used to construct data structures which are used during the CPP's operation to validate it's activity. While the current white list data is merely text, it is possible to imagine easy and effective means of cryptographically protecting it from surrepticious modification.

White List Validation: CuPIDS currently uses the function call entry/exit events to perform white-list validation of control flow within the CPP. Each event contains a source and destination address pair for each function that is called, and these pairs are compared by the CSP to allowed pairs. The same is true for library calls, intra-library calls, and system calls. Any malfunction in the CPP process which leads to code being run in a pattern that is not in the white list will be caught by the CSP. In the case where the code makes a call to a function it is not allowed to call, the illegitimate function entry event will be caught by the CSP. If the call into the function bypasses the prologue code and an entry event is not generated, the exit event will be caught by the CSP.

4.1.6. Stack Monitoring CuPIDS uses function entry and exit events to model the CPP's program stack. Events that break the stack model legitimately can be added to the white list, either manually or automatically during training sessions. Examples of exceptional but legitimate events may include setjmp/longjmp pairs and exception or signal handlers.

4.1.7. Enabled/Disabled Events It may not be possible/desirable to enable all possible events. Therefore it may be useful to allow the CSP to enable or disable specific events or blocks of events at runtime. Advantages to doing this include the ability to tune the level of inspection based upon factors such as timing (beginning of software operation versus after it has been running for a while), external factors such as newly discovered threats, and increased attacker uncertainty (randomly turning on events makes it more difficult for the attacker to work safely).A possible mechanism that would allow this behavior is creating a fully instrumented binary, constructing in the CSP's inputs a map of all the event generators, and then disabling the unneeded ones by writing nops into the text in place of the calls to the function entry/exit routines. To re-enable the event the CSP can write the appropriate instructions back into the text.

4.1.8. Spoofing Protection To protect against an attacker spoofing the interrupt-based IPC mechanism handlers cap-

ture the return address of the caller from the trap frame. Thus, a spoofed call coming from an illegitimate location will be caught.

4.1.9. Forensics When CuPIDS detects problems in a CPP it can freeze that process, write its entire state out to disk, and start up a new instance of the program. The saved state can include the normal core dump, the kernel structures related to the process, the state of any files in use by the process at the time of error, and the runtime history stored in the CSP. This data will allow for complete analysis of the fault.

5. Results

We have used this prototype to verify basic CuPIDS functionality. The system is able to correctly load and execute CPP and CSP components, the CSP is able to detect invariant and security policy violations as well as illegitimate control flow changes. Upon detecting a fault or attack, the CSP is able to halt the PP, raise an alarm, save the state of the CPP's memory and execution trace history, and in some cases repair the damage from the attack or error, allowing the CPP to continue execution without interruption. Time-related testing results are discussed below.

The experiments described here demonstrate that it is possible for one process to efficiently perform realtime runtime error checking on variables in another process as well as perform simple flow control validation. To demonstrate the validity of our research hypothesis we demonstrate that CuPIDS can provide guaranteed detection of certain attacks before a context switching event occurs. This claim cannot be matched by a StUPIDS, even if equipped with a comparable detector set.

In our experimentation we used a combination of widely-used, open source applications and servers as well as applications created specifically to test certain aspects of CuPIDS' functionality. The commonly used applications were WU-FTP version 2.6.2 and gnats version 3.113.12. These programs were chosen because they represent software typical of that used in our target environment, their source code is available so that we could examine and instrument them, and because they contain exploitable vulnerabilities as demonstrated by publicly available zero-day exploits.

WU-FTP's ftpd daemon was used to perform performance measurements of CuPIDS as well as to test CuPIDS' invariant violation detection and self-healing capabilities. The ftpd daemon was ideal for this purpose because it is fairly large (about 20,000 lines of code), its behavior is representative of many server-type applications in that it runs for long periods and forks off child processes to handle requests, and finally because it has a number of buffer overflow vulnerabilities[5].

We used gnats-3.113.12 because of the existence of a locally exploitable vulnerability[6]. gnats was used to test CuPIDS' ability to detect invariant violations.

We used the CuPIDS prototype to perform a number of experiments, allowing us to demonstrate the validity of our research hypothesis. A detailed discussion of those results is available in [citation blinded] , and summarized here.

5.1. Test Platform

The experiments described below were run on a MP platform with dual Xeon 2.2GHz processors, 1G RAM, 1 120GB ATA100 drive. Hyperthreading (HTT) was enabled so the operating system had available 4 CPUs. We recognize that the performance of HTT processors does not match that of separate CPUs[18]; however, the architecture is useful to us for other reasons. While the results discussed here do not make use of HTT specific features we do make use of the fact that they share architectural components as discussed in [citation blinded] . For experiments involving CuPIDS, the CSP is the only user of CPU1, the instrumented ftpd uses all of one CPU's cycles, and the ftp client uses all of another CPU's cycles, and the system, including the test drivers, mostly run on the fourth CPU. The test drivers ensure that all file I/O is done on local drives so that network overhead does not become a factor. During the non-instrumented experiments CPU1 is held idle to provide ftpd the same operating environment as it had in the instrumented runs. ftpd was run as root in standalone mode (command line `ftpd -s`, which causes it to stay in the foreground and fork processes as needed).

5.2. Runtime Efficiency Tests using WU-FTP

The initial experiments connect to the ftp daemon, log in, and perform 300 ftp file transfers and one ls for a total of 301 transfers. The file transfer workload is 1,881,832,400 bytes and the overall workload per experiment is 1,881,904,317 bytes. Three sets of five experimental runs were made, one using the CuPIDS interrupt-based IPC, one using SysV IPC, and one baseline test was run against a non-instrumented version of WU-FTP. The results are summarized in Table 2.

The initial tests are intended to measure the overhead involved in getting CuPIDS events out of the CPP and into the CSP, therefore we constructed a worst-case event load based on program flow control monitoring. In the instrumented tests, all function calls generate entry and exit events. This includes internal functions, libc and intra-libc

5 CVE entries CVE-1999-0878, CAN-2003-0466, and CVE-1999-0368 [22]

6 CVE CAN-2004-0623[22]

Event Communications Method	Real Time (seconds)	Throughput (MB/seconds)
Interrupt-based mean	130.02	14.97
Interrupt-based stdev	0.52	0.05
SysV IPC-based mean	241.38	13.77
SysV IPC-based stdev	2.19	0.11
Non-Instrumented mean	118.50	16.16
Non-Instrumented stdev	0.10	0.02

Table 1. WU-FTP Runtime Communication Performance Measurements

calls as well as system calls. Each event includes caller address and callee address information. These events are validated against a white list of calls statically extracted from the ftpd binary. The initial white list contained all the legitimate non-function-pointer-based function and shared library calls as well as a list of all function pointer uses. An initial experimental run identical to the timing runs was made to train the CSP on the actual function pointer usage. The CSP received each function/library/system call event, verified it against the white list, and used it to model the CSP's program stack. The timing related tests did not include embedded invariant tests.

Each experimental run took between two and four minutes and generated approximately 1.4 million events corresponding to WU-FTP's activities. As shown in Table 2 the overhead of generating and using those events was less than ten percent for the CuPIDS IPC as opposed to approximately 100 percent for the SysV-based IPC. Note that this overhead should be balanced against the removal of an inline IDS doing the same tasks. Even a standalone IDS with a similar detector set would be competing for CPU cycles with the CPP, likely degrading application performance.

5.3. Control Flow Change Results

A number of experiments were run to validate CuPIDS' ability to detect illegitimate control flows in the CPP. A summary of those experiments is presented here and described in more detail in [citation blinded] .

- Illegitimate Syscall Detection: Both gnats and WU-FTP were used in these tests. In both applications a buffer was overflowed in such a way that bytecode contained in the overflow string was executed. The injected code made a number of system calls from the stack. CuPIDS was able to detect all of the illegitimate system calls.
- Illegitimate Internal Function Call Detection: Both gnats and WU-FTP were used in these tests. In both applications CuPIDS was able to detect an internal function call that had been removed from the white list (simulating the activity of injected code that makes calls to functionality embedded in the vulnerable application). CuPIDS was also able to detect calls into functions that bypassed the prologue event generator. It did so by detecting illegitimate program stack activity in the stack model.
- Illegitimate Library Call Detection: Both gnats and WU-FTP were used in these tests. In both applications CuPIDS was able to catch a call to a library function which was removed from the white list.
- Spoofing/Masquerading Detection: CuPIDS detected attempts to make library or system calls from locations other than those specified in the white lists. This prevents attackers from performing masquerading attacks such as those described in [23]. The CuPIDS IPC mechanism guards against spoofed event generation by including in each event the return address for the generating function as taken from the stack. As the address is placed on the stack by the processor and reading it occurs in kernel space there is no way for a user program to spoof this information.
- Direct Variable Protection: WU-FTP was used for these experiments, which involved performing invariant testing on simple variables (int, char, simple structs) and a string buffer. As discussed earlier, CuPIDS was able to detect illegitimate changes to both classes of variables. In the case of a stack-based buffer overflow it was able to detect the overflow, save the overflowing data, repair the corruption to memory following the buffer, terminate the string in the buffer appropriately (by writing a zero into the end of the buffer), allow the CPP to continue running, and write the overflow string and information about the overflow out to disk. In these experiments the detection took place as the overflow occurred, so CuPIDS was able to halt the CPP before it could return into the corrupted instruction pointer on the stack. Therefore the attack was stopped before any control flow change took place—a capability unique to a parallel monitoring architecture such as CuPIDS. Even had the buffer overflow not been directly detected, CuPIDS would have detected the control flow change to the stack and may have been able to make the same repair.

5.4. Time to Detect

We ran a number of experiments to determine how quickly CuPIDS detected illegitimate events. Two types of tests were run: one which performed an invariant

test upon notification that a variable access was complete, and one in which real-time monitoring was used. Measuring the detect times for these tests without a hardware-based in-circuit emulator (ICE) proved challenging. Our theory stated that CuPIDS' ability to perform simultaneous monitoring of memory shared using the virtual memory mapping capability would result in detection at the point the invariant was violated[7]. Using the O/S clocks to mark violation and detection times was not feasible because of the overwhelmingly large overhead imposed by system calls. To quantify how quickly the CSP detects a problem we instrumented the CPP by adding counter that starts incrementing immediately following the completion of a monitored variable access. When the CSP detects a violation it immediately takes a snapshot of this counter. A buffer overflow in WU-FTP was used as the invariant violation. Each set of tests was run in both CuPIDS multi-processor (MP) mode and StUPIDS uni-processor (UP) mode. The results of these experiments are as follows:

Simultaneous Monitoring In these tests upon notification that a protected variable is to be accessed, a monitoring task is started. In the CuPIDS case this monitor is placed on the CuPIDS CPU and runs alongside the CPP. In the StUPIDS case the monitor is scheduled as is any other task and its execution is interleaved with the execution of the CPP. The average of 40 million instructions executed by the UP CPP before overflow detection takes place compared to the immediate detection of the overflow in the CuPIDS CPP validates our research theory—that architectures such as CuPIDS can detect illegitimate events faster than can UP architectures.

Blocking Invariant Checking In these tests, the CPP sends a blocking checkpoint event to the CSP immediately following the variable access. to accessing the protected variable. Because the CPP is not allowed to continue execution until the invariant test is complete it is not surprising that both MP and UP mechanisms immediately caught the overflow.

Non-blocking Invariant Checking In these tests, the CPP sends a non-blocking checkpoint event to the CSP immediately following the variable access and continues execution. The consistant results from the CuPIDS CSP are expected, and reflect the amount of time it takes to perform the invariant test. The much higher and inconsistent results from the UP CSP reflect the scheduler-based non-determinism faced by all StUPIDS architectures.

7 Actually, at the point the cache snooping mechanism detected the shared usage of the memory location and propagated the change from the CPP's CPU into main memory and the CSP's CPU's cache.

Monitor Type		MP Mode (# instr)	UP Mode (# instr)
Simultaneous Monitoring	mean	immediate	$40.72 * 10^6$
	stdev	0.0	$0.76 * 10^6$
Blk Invariant Testing	mean	immediate	immediate
	stdev	0.0	0.0
Non-blk Invariant Testing	mean	$52.07 * 10^3$	$3.94 * 10^6$
	stdev	$18.63 * 10^3$	$10.81 * 10^6$

Table 2. WU-FTP Buffer Overflow Detect Measurements

6. Future Work

The architecture described is the initial work to achieve the broad goals of CuPIDS. The prototype is designed to serve as a platform for future research and development in areas of intrusion detection, forensics, and other security related tasks. In addition to completing an implementation of the CuPIDS architecture described in this paper and testing our research hypothesis there are a number of related avenues we intend to explore. These include overall self-protection and healing to enhance fault tolerance as well as support for computer forensics.

6.1. O/S Self-protection

The growing body of work into mandatory access control (MAC) mechanisms such as those based on Biba's integrity-based [24], and Bell and LaPadula's multi-level security [25] models are used to provide a first-line defense against user application compromise. While MAC protection systems are not novel, the CuPIDS architecture uses hardware protection mechanisms in commodity CPUs to define and protect the MAC mechanism and CuPIDS themselves against direct attacks that attempt to bypass its controls.

7. Conclusion

We have proposed a paradigm shift in computer security, one that challenges conventional wisdom by trading performance for security. Our approach is based upon running dedicated monitoring functions parallel with the code they monitor on a MP system. We believe the CuPIDS architecture to be more effective than StUPIDS architectures in terms of real-time detection of bad events as well as offering some novel detection techniques based upon the low-level

and parallel nature of the monitoring. By dedicating computational resources explicitly to security tasks we are trading performance for security; however, by offloading some security tasks from the production process into the security process and running them in parallel we are decreasing the workload of the system production components. We have constructed a prototype of this architecture and used it to not only validate CuPIDS' basic functionality, but more importantly verify that such systems are likely to be psychologically acceptable in terms of performance and ease of development.

References

[1] B. A. Kuperman, *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources*. PhD thesis, Purdue University, West Lafayette, IN, 08 2004. CERIAS TR 2004-26.

[2] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, September 1975.

[3] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Tech. Rep. 99-15, Chalmers Univ., Mar. 2000.

[4] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in C programs," *Softw. Pract. Exper.*, vol. 27, no. 1, pp. 87–110, 1997.

[5] H. Xu, W. Du, and S. J. Chapin, "Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths," 2004. In Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection.

[6] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, p. 62, IEEE Computer Society, 2003.

[7] R. Gopalakrishna, E. H. Spafford, and J. Vitek, "Efficient intrusion detection using automaton inlining," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2005.

[8] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *SP '01: Proceedings of the IEEE Symposium on Security and Privacy*, p. 156, IEEE Computer Society, 2001.

[9] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, "Formalizing sensitivity in static analysis for intrusion detection.," in *IEEE Symposium on Security and Privacy*, 2004.

[10] J. D. Tygar and B. Yee, "Dyad: A system for using physically secure coprocessors," in *IP Workshop Proceedings*, 1994.

[11] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *In Proceedings 1997 IEEE Symposium on Security and Privacy, pages 65–71, May 1997.*, 1997.

[12] O. S. Saydjari, "LOCK: An Historical Perspective," in *Proceedings of the 18th Annual Computer Security Applications Conference, 2000*, (www.acsac.org), pp. Online, www.acsac.org, ACSAC, 2000.

[13] X. Zhang, L. van Doom, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection," in *ACM European SIGOPS 2002*, 2002.

[14] J. Molina and W. A. Arbaugh, "Using independent auditors as intrusion detection systems," in *Proceedings of the Fourth International Conference on Information and Communications Security (S. Qing, F. Bao, and J. Zhou, eds.)*, vol. 2513 of LNCS, pp. 291–302, 2002.

[15] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[16] J. P. Anderson, "Computer security technology planning study," Tech. Rep. ESD-TR-73-51, Vol. II, HQ Electronic Systems Division (AFSC), Hanscom Field, Bedford, MA, 01730, 1972.

[17] R. Lipton, S. Rajagopalan, and D. Serpanos, "Spy: A method to secure clients for network services," in *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, 2002.

[18] I. Tm-I, "Ia-32 Intel architecture software developers manual volume 1: Basic architecture."

[19] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons, Inc., 2001.

[20] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*. Pearson Higher Education, 2004.

[21] TrustedBSD, "TrustedBSD." www.freebsd.org.

[22] Mitre, "Common vulnerabilities and exposures.."

[23] T. Ptacek and T. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," tech. rep., Secure Networks, Inc., 1998.

[24] K. Biba, "Integrity considerations for secure computer systems," Tech. Rep. TR-3153, Mitre, Bedford, MA, Apr. 1977.

[25] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations and model," Tech. Rep. M74-244, The MITRE Corp., Bedford MA, May 1973.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	5.Jul.05	MAJOR REPORT

4. TITLE AND SUBTITLE
AN IMPLEMENTATION OF CUPIDS: EVALUATING THE EFFECTIVENESS OF MULTI-PROCESSOR INFORMATION SYSTEM SECURITY.

5. FUNDING NUMBERS

6. AUTHOR(S)
CAPT WILLIAMS PAUL D

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
PURDUE UNIVERSITY

8. PERFORMING ORGANIZATION REPORT NUMBER
CI04-1128

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS

15. NUMBER OF PAGES
11

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

THE VIEWS EXPRESSED IN THIS ARTICLE ARE THOSE OF THE AUTHOR AND DO NOT REFLECT THE OFFICIAL POLICY OR POSITION OF THE UNITED STATES AIR FORCE, DEPARTMENT OF DEFENSE, OR THE U.S. GOVERNMENT.